Reusable Business Tier Architecture Driven by a Wide Typed Service

Óscar Mortágua Pereira[1], Rui L. Aguiar[2]
Instituto de Telecomunicações
DETI, University of Aveiro
Aveiro, Portugal
{omp[1],ruilaa[2]}@ua.pt

Maribel Yasmina santos
Centro Algoritmi
DSI, University of Minho
Guimarães, Portugal
maribel@dsi.uminho.pt

***Abstract*— Call Level Interfaces (CLI) are difficult to use mainly in intensive database applications with many Create, Read, Update and Delete (CRUD) expressions. As low level API, they are not suited to promote the development process of business tiers as reusable components, leading to the need of writing additional source code whenever a new CRUD expression is needed. To tackle this gap, this paper proposes an architecture for building reusable business tier components herein referred to as Reusable Business Tier Architecture (RBTA). It relies on a single customizable wide typed service to address a business area, such as accounting. The typed service is able to manage all the required CRUD expressions for that business are, which are deployed at runtime in accordance with the user's needs. The only constraint is that the required service to manage each CRUD expression must be a sub-set of the implemented wide typed service. A proof of concept based on Java Database Connectivity (JDBC) is also presented.**

Keywords-component; softwae reuse, software architecture, business tiers, call level interfaces, relational databases.

I. INTRODUCTION *(HEADING 1)*

Object-oriented and relational paradigms are simply too different to bridge seamlessly, leading to a set of difficulties informally known as *impedance mismatch* [1]. Impedance mismatch derives from the diverse foundations of both paradigms and has been an open issue for more than 50 years [2]. To tackle impedance mismatch, several solutions have been proposed, including Call Level Interfaces (CLI), Embedded SQL, object-to-relational mapping techniques (O/RM), language extensions and persistent frameworks. These solutions were designed to deal with and hide all the complexity of the translation process between the two paradigms. Among the available solutions, CLI are considered the correct option whenever a fine tune control on the interactions with the host database is a key requirement [2]. In spite of this important advantage, CLI were not designed to build reusable business tier components leading to the following drawbacks: 1) the same Create, Read, Update and Update (CRUD) expression is frequently re-written to address different business needs; 2) whenever a new CRUD expression is needed there is no option than write the necessary source-code to manage its execution; 3) CLI do not provide any architecture to decouple the development process of business tiers from the development process of application tiers. These drawbacks are increasingly relevant when schemas of databases get more and more complex, when the number of CRUD expressions increases and when the complexity of CRUD expressions increases. In fact, CLI are general low level API that do not provide any architectural policy to be followed during the development process of business tiers to address any of the mentioned drawbacks. They were mainly designed to address the impedance mismatch issue. Figure 1 presents a simple CRUD expression using a CLI, in this case JDBC [3]. It depicts a program to retrieve data from a list of products and to update the attribute *unitPrice* kept in a table named *Products*. The list of products to be retrieved and updated is included in *List<Integer> productId* and the new values for attribute *unitPrice* are included in *List<Float> unitPrice* (see arguments of method *updUnitPrice*). The CRUD expression has to be written (line 46), prepared (line 47-49,51) and executed (line 52). To access the returned attributes, programmers need to master the schema of the returned relation (line 54-56, 58). Moreover, if a new case using the same CRUD expression is needed to update another attribute (*UnitsInStock*), see Figure 2, it would be necessary to re-write identical source-code for the business tier (including the same CRUD expression) and master the same sub-schema. CLI do not provide any means to avoid the need of mastering database schemas, to prevent the need for re-writing the same CRUD expressions and to prevent the need of writing repeated source-code for business tiers. Another relevant aspect is the tangled source-code of business and application tiers, which prevents software designers to decouple the development process of both tiers. In Figure 1, business source-code (line 46-49, 54-56,58) co-exists with application source-code (line 43-45, 50-60), which is also extensible to Figure 2. These difficulties are particularly

```
43  void updUnitPrice(List<Integer> productId,
44                    List<Float> unitPrice)
45                    throws SQLException {
46      sql="Select * from Products where productID=?;";
47      ps=conn.prepareStatement(sql,
48                          ResultSet.TYPE_FORWARD_ONLY,
49                          ResultSet.CONCUR_UPDATABLE);
50      for (int n=0;n<productId.size();n++) {
51          ps.setInt(1,productId.get(n));
52          rs=ps.executeQuery();
53          if (rs.next()) {
54              pn=rs.getString("ProductName");
55              supID=rs.getInt("SupllierID");
56              // ... more code
57              uPrice=unitPrice.get(n);
58              rs.updateFloat("UnitPrice",uPrice);
59              rs.updateRow();
60              // ... more code
```

Figure 1. Usage of CLI (JDBC)

978-1-4799-0172-2/13 $26.00 copyright 2013 IEEE
ICIS 2013, June 16-20, 2013, Niigata, Japan

```
65  void updUnitsInStock(List<Integer> productId,
66                       List<Float> units)
67                       throws SQLException {
68      sql="Select * from Products where productID=?;";
69      ps=conn.prepareStatement(sql,
70                               ResultSet.TYPE_FORWARD_ONLY,
71                               ResultSet.CONCUR_UPDATABLE);
72      for (int n=0;n<productId.size();n++) {
73          ps.setInt(1,productId.get(n));
74          rs=ps.executeQuery();
75          if (rs.next()) {
76              pn=rs.getString("ProductName");
77              supID=rs.getInt("SupllierID");
78              // ... more code
79              uInStock=units.get(n);
80              rs.updateFloat("UnitsInStock",uInStock);
81              rs.updateRow();
82              // ... more code
```

Figure 2. Duplication of source code with CLI.

relevant when databases have complex schemas and when business tiers have many and complex CRUD expressions. To tackle these CLI drawbacks, a research has been carried out in the context of Component-Based Software Engineering [4, 5]. Component-based development aims to compose software units from other pre-built software units [4]. At the end, a final software system is not built as a unique block but as a composite of software units known as components [6]. A key aspect for the success of any component is its capability of being adapted to be reused [7] which is improved if another key aspect is also considered: the reuse of computation [8]. Thus, this research is focused on creating an software architecture for building reusable business tier components, herein referred to as the Reusable Business Tier Architecture (RBTA). The adaptation process is carried out in two phases: the static phase and the dynamic phase. During the static phase, reusable business tier components (RBTC) are automatically built from a wide typed service derived from a business schema. The business schema fully addresses a business area, such as accounting, warehouse or sales. During the dynamic phase, CRUD expressions are deployed, at runtime, to address specific users' needs. The only constraint is that the required service to manage each individual CRUD expression must be a sub-set of the implemented wide typed service. In this paper a proof of concept is also presented. Throughout this paper, all examples are based on Java, Java Database Connectivity (JDBC) [3], T-SQL (SQL Server) and Northwind Microsoft database [9]. The presented code may not execute properly since we only show the relevant parts for the points under discussion.

This paper is structured as follows: section II presents an overview of the related work; section III introduces Call Level Interfaces; section IV presents the Reusable Business Tier Architecture; section V presents a proof of concept and, finally, section VI concludes and presents future work.

II. Related Work

O/RM tools were devised to create in the object-oriented paradigm static representation models of relational database schemas. The static model is built in a first stage, eventually by a database administrator, and then programmers start the development process. The basic units of the static representation models are classes (entities), each one representing a database table. Through these entities programmers may read data from tables, update data, insert new data and, finally, delete existing data. To support explicit CRUD expressions, O/RM tools provide language extensions and proprietary SQL languages. Despite these advantages, O/RM present several drawbacks: 1) They induce an additional overhead when compared to CLI [2]; 2) O/RM tools support but they were not devised to address complex CRUD expressions; 3) O/RM tools provide a set of extended functionalities (support for native and proprietary SQL languages, and language extensions) promoting the tangling of source-code of the business tiers and application tiers; 4) the context in which the extended functionalities of O/RM tools are used does not promote their reuse in different business needs. In spite of these drawbacks, O/RM are powerful tools when the key aspect is restricted to the translation of database tables into object-oriented entities (typed-objects).

Safe Query Objects [10] combine object-relational mapping with object-oriented languages to specify queries using strongly-typed objects and methods, relieving programmers from writing traditional CRUD expressions. They rely on Java Data Objects [11] to provide strongly-typed objects and also to provide data persistence. Safe Query Objects are a promising technique to express queries but they also convey the presented CLI drawbacks. The only exception is the need for writing CRUD expressions. Moreover, although joins can be used for filters, the result is always of a single typed object - there is no possibility to project more than one table. This constraint definitely prevents the use of Safe Query Objects. The paper does not present any performance assessment but as it uses Java Data Objects [11] we may foresee a lower performance than CLI.

SQL DOM [12] generates a Dynamic Link Library containing classes that are strongly-typed to a database schema. These classes are used to construct dynamic CRUD expressions without manipulating any strings. Similarly to Safe Query Objects, SQL DOM does not tackle CLI drawbacks and its performance exhibits very poor results.

Aspect-oriented programming [13] community considers persistence as a crosscutting concern [14]. Several works have been presented but none addresses the points here under consideration. The following works are emphasized: [15] is focused on separating scattered and tangled code in advanced transaction management; [14] addresses persistence relying on AspectJ; [16] presents AO4Sql as an aspect-oriented extension for SQL aimed at addressing logging, profiling and runtime schema evolution. It would be interesting to see an aspect-oriented approach for the points herein under discussion.

III. Call Level Interfaces

This section briefly introduces the main functionalities of CLI. Their presentation is important to provide some background to understand RBTA functionalities that are directly derived from CLI.

JDBC [3], ODBC [17] and ADO.NET [18] are three representative of CLI. CLI provide a wide set of

functionalities but only those directly related to the execution of CRUD expressions will be considered. Services such as those for managing connections to host databases are out of the scope of this research and not addressed in this paper.

CRUD expressions are executed against the host database and the possible results they produce (only for Select expressions) are locally managed by local memory structures (LMS) – (ResultSet [19] for JDBC, RecordSet [20] for ODBC, DataSet [21] for ADO.NET). Associated with LMS there are two common relational concepts: Relation is a data structure returned from a database schema when a Select is executed. Tuple is a row of one relation. Figure 3 presents a general LMS containing 5 tuples (1 to 5) and 6 attributes (a, b, c, d, e, f). This LMS could have been instantiated to manage the data returned by the following CRUD expression: *Select a, b, c, d, e, f from Table Where ….* In this case, the CRUD expression has returned 5 tuples and the current selected tuple is row number 2.

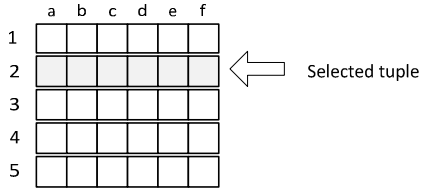

Figure 3. LMS with 5 tuples and 6 attributes.

Key services of CLI are organized in three main categories: execution target, scrollability and updatability.

Execution target: it comprises services related to the execution of CRUD expressions. CRUD expressions are executed as compiled-on-the-fly or pre-compiled (when they are to be reused or when they have parameters defined at runtime). Additionally, CLI deal differently with Select expressions from the other three types of CRUD expressions. Select expressions instantiate an LMS while the other types do not. The latter group generates a value for the number of affected rows in the host database.

Scrollabilty: it comprises services related to the scrolling process on LMS. There are two mutual-exclusive possibilities defined at LMS instantiation time: *forward-only* – in this case it is only possible to move forward one tuple at a time; *scrollable* – in this case it is possible to move in any direction and jump several tuples at a time.

Updatability: it comprises services organized in protocols to interact with data contained in LMS. There are two mutual-exclusive possibilities defined at LMS instantiation time: *read*-only – the content of the LMS is read-only and no changes are allowed; *updatable* – changes may be performed on LMS (insert new tuples, update tuples and delete tuples). CLI commit these changes into the host database.

IV. RBTA

This section is organized as follows: firstly, Business Tier Interface, which is a key artifact of RBTA, is introduced and presented and, only then RBTA is presented.

A. Business Tier Interface

The RBTA needs to provide a wide typed service to address the needs of a business area. The service may be defined as an interface which is herein known as Business Tier Interface (BTI). BTI accepts and manages the execution of CRUD expressions, deployed at runtime, on behalf of application tiers. There are four types of CRUD expressions (Insert, Select, Update and Delete) each one with its own requirements. Therefore, Business Tier Interface needs to support the four types of CRUD expressions, each one specialized to manage one of the types. One possibility to address this specialization is through the definition of one interface for each type of CRUD expressions. This revealed to be a good approach because in spite of having four specialized interfaces each one could be built from a pool of smaller interfaces herein known as Low Level Interfaces. Basically, each Low Level Interface comprises and organizes a set of coherent and related functionalities of CLI. Then, each specialized interface, herein known as High Level Interface, is built by gathering some of the Low Level Interfaces.

Low Level Interfaces

Figure 4 shows the six low level interfaces: IExecute, IResult, IRead, IWrite, IControl and ISet. Next follows a detailed description for each low level interface.

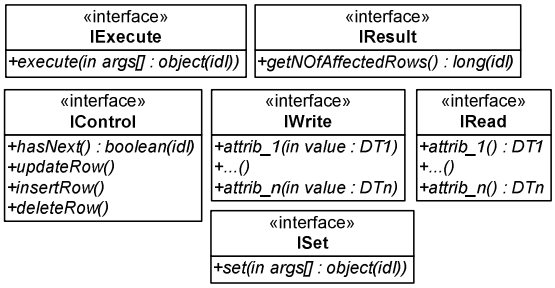

Figure 4. Low level interfaces.

execute: this method executes the underlying CRUD expression.

returns: void.

args: It is used to define the runtime values for clause conditions of CRUD expressions. It is not possible to foresee all the possibilities for the clause conditions for all CRUD expressions. One possibility to overcome this issue is to define one argument able to support any number of values and of any data type. The chosen approach uses a single array of objects to support any number of clause conditions defined at runtime. Next, an example is shown. The execution of the following Select expression, *Select * From Table Where id=@id and value > @value* from the application tier point of view, in Java should have the implementation shown in Figure 5.

```
s.execute(new Object[]{id,value});
```

Figure 5. Example of execute usage.

IResult interface is associated with Insert, Update and Delete CRUD expressions and it is used to collect the number of effected rows as consequence of their execution.

getNAffectedRows: this method returns the number of affected rows.
returns: number of affected rows.

ISet interface is used to set the runtime values for the attributes used on all Insert and Update expressions.

set: this method sets the runtime values for the attributes used on all Insert and Update expressions. Although each CRUD expression may have its particular attributes, this method has an independent signature as the one used with the *execute* method.
returns: void.
args: It is a single array of objects to support any number of values defined at runtime. The case presented for the implementation and for the use of the *execute* method is directly applied to this case.

IRead interface usage is restricted to Select expressions. It is used to read attributes from LMS and it comprises as many getter methods as the possible different attributes of all returned relations. Each method's signature is strictly related to a returned attribute. The presented interface, see Figure 4, suggests that attributes *attrib_1*, …, *attrib_n* are returned from the host database. Their correspondent data type in the host programming language are *DT1*,…,*DTn*, respectively. As an example, the two next Select expressions are supported by the IRead interface if their SQL data types are in accordance with the ones defined in the IRead interface. With a single IRead interface we open the possibility to read attributes from LMS filled with different Select expressions. The only constraint is that the schema of the returned relation must be contained the implemented IRead interface.

```
Select attrib_1,attrib_3,attrib_7 from ...
Select attrib_10, attrib_1, attrib_30 from ...
```

IWrite interface usage is restricted to Select expressions. It is used to write values into attributes of LMS and it comprises as many setter methods as the possible different updatable attributes of all returned relations. Each method's signature is strictly related to a returned attribute. The presented IWrite interface in Figure 4 suggests that attributes *attrib_1*,…, *attrib_n* are of data type *DT1*,…,*DTn*, respectively. Similarly to IRead, a single IWrite interface opens the possibility to write on updatable attributes of LMS filled with different Select expressions.

IControl interface usage is restricted to Select expressions. It may comprise other methods depending on the particular implementation. We have only shown some of the most important methods:

hasNext: this method moves the cursor one tuple forward and points to the next available tuple.
returns: boolean (*true* – there is next tuple, *false* – otherwise)
updateRow: this method commits changes previously made through *IWrite* interface.
returns: void.
insertRow: this method commits a new row previously inserted through the *IWrite* interface.
returns: void.
deleteRow: this method deletes the current selected row.
returns: void.

From the six low-level interfaces, only IRead and IWrite are customizable to address each business area or, in other words, each RBTC. They depend on the schema of all possible returned relations. All other low-level interfaces are immutable and, therefore, are shared by all RBTC. This is an important issue because it has a significant impact on minimizing the necessary effort to accomplish the static adaptation process of RBTC.

High level interfaces

The four High Level Interfaces are presented in Figure 6, one for each type of CRUD expression: ISelect, IInsert, IUpdate and IDelete. Next, we will thoroughly explain the composition of each high-level interface.

ISelect: Select expressions are managed by the ISelect interface which comprises four low-level interfaces: 1) IExecute to execute Select expressions and also for setting the runtime values of their clause conditions, 2) IRead to read the attributes of the returned relations and 3) IWrite to change the contents of returned relations, and 4) IControl to control some actions on the returned relations, such as commit changes into databases and the scrolling process.

IInsert: Insert expressions are managed by the IInsert interface which comprises three low-level interfaces: 1) IExecute to execute Insert expressions and 2) ISet to set the runtime values for the column list and 3) IResult to get the number of modified rows in the host table as consequence of the CRUD execution.

IUpdate: Update expressions are managed by the IUpdate interface which comprises the same low-level interfaces as IInsert.

IDelete: Delete expressions are managed by the IDelete interface which comprises two low-level interfaces: 1) IExecute to execute Delete expressions and to set the runtime values of their clause conditions and 2) IResult to get the number of deleted rows in the host table as consequence of the CRUD execution. Thus, the four facets of Business Tier Interfaces are: ISelect, IInsert, IUpdate and IDelete.

Each High-Level interface is associated with a CRUD expression type and for each one, one Business Tier Entity

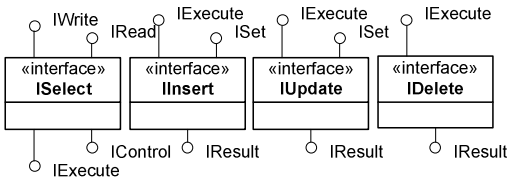

Figure 6. The four high level interfaces.

(BTE) is defined. A Business Tier Entity is a software unit responsible for the implementation of one of the High-Level interface. An instance of a Business Tier Entity is herein known as Business Tier Worker (BTW). Additional details are provided in the next sub-section.

B. RBTA Presentation

Beyond the previous interfaces the RBTA needs additional services to be able to promote the development of reusable components. Therefore, we start to introduce IConfig, IUser and ISession, see Figure 7. Then, the final architecture is presented.

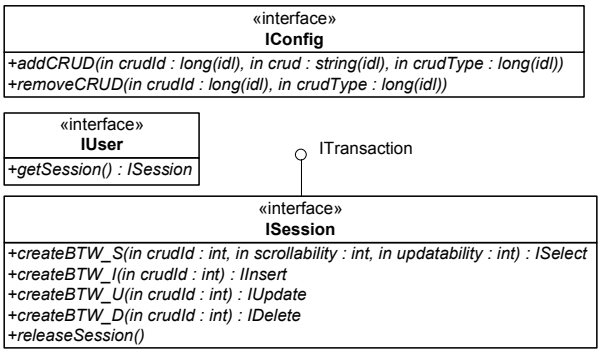

Figure 7. IConfig, IUser and ISession interfaces.

IConfig interface is used to deploy CRUD expressions at runtime. This possibility allows RBTC to be adapted at runtime to cope with different needs of different users. Each CRUD expression is attached (addCRUD) and detached (removeCRUD) to a type of Business Tier Entity identified by the argument *crudType*. Each CRUD expression (*crud*) is uniquely identified by its id (*crudId*) for each Business Tier Entity type. This way, each user is able to use and execute the set of CRUD expressions that address its specific needs.

IUser interface is used to create sessions. A session owns a private connection to the host database.

ISession interface is the factory of RBTA and it is used by application tiers to create Business Tier Workers. There is one method for each type of Business Tier Entity. *createBTW_S* has two additional arguments to allow a full control on the functionalities to be provided by LMS. Additionally, it extends an *ITransaction* interface for managing database transactions.

createBTW_S: this method creates a Business Tier Worker instance of type Select.
returns: ISelect.
crudId: CRUD expression to be executed.
createBTW_I: this method creates a Business Tier Worker instance of type Insert.
returns: IInsert.
crudId: CRUD expression to be executed.
createBTW_U: this method creates a Business Tier Worker instance of type Update.
returns: IUpdate.
crudId: CRUD expression to be executed.
createBTW_D: this method creates a Business Tier Worker instance of type Delete.
returns: IDelete.
crudId: CRUD expression to be executed.

ITransaction is not described but it comprises the traditional services used to manage database transactions, such as: begin transaction, commit transaction and rollback transaction.

Figure 8 presents a class diagram for RBTA. The entry point is the public static method *createRBTC* of *RBTC* which creates a new instance of an RBTC and returns IRBTC interface. The arguments are not shown but at least the information to establish a connection to a database is necessary. From IRBTC interface, users access RBTC functionalities: IConfig to configure RBTC and IUser to create new sessions. Each type of CRUD expression is managed by a different type of Business Tier Entity: BTE_S for Select, BTE_I for Insert, BTE_U for Update and BTE_D for Delete CRUD expressions.

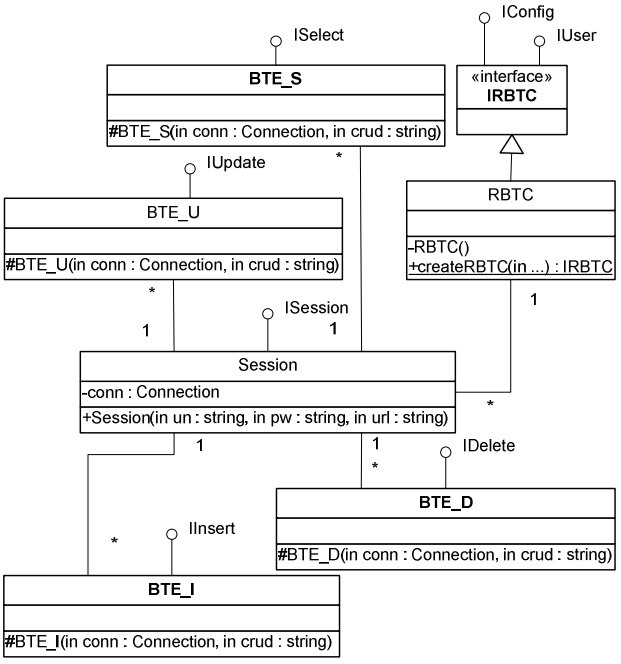

Figure 8. Class diagram of RBTA.

V. PROFF OF CONCEPT

A proof of concept based on Java and JDBC (sqljdbc4) for SQL Server 2008 was developed. Tests have been concluded and a trial version may be downloaded from here[1]. A new component was built: Business Tier Component Builder (BTC Builder). BTC Builder is used to automatically create and compile the source code for components based on RBTA. The biggest challenge was centered on the approach to be followed to formalize IRead and IWrite interfaces. Several approaches were considered, among them XML and standard Java interfaces. In spite of being less expressive than XML, Java interfaces proved to be an efficient and effective approach. Programmers do not need to use a different development environment, interfaces are basic entities of any object-oriented programming language and are widely used, interfaces are easily edited and maintained and, finally, IRead and IWrite have also been defined as interfaces. These were the fundamental reasons for having opted for Java interfaces in detriment of XML. IRead and IWrite are the only low-level interfaces that vary from component to component. In spite of being two different interfaces, one of them is easily inferred if the other one is known. Figure 9 and Figure 10 partially presents IRead and IWrite interfaces used in the scenario herein implemented (tables and attributes were renamed to avoid duplicated attribute names). The signature of each method is mandatorily based on the correspondent attribute schema: name and data type. If an attribute *attrib* is of data type *varchar* then the getter and setter methods are *String attrib()* and *void attrib(String att)*, respectively. Thus, to build a component derived from RBTA, it is only necessary to write one interface and then, through a tool, automatically create its source-code. In our case, BTC Builder uses reflection on IRead interfaces to build RBTC.

A scenario was defined and created to evaluate the RBTA. Two Select expressions were written to project all attributes of two tables: *Categories* (s1) and *Products* (s2). For the same two tables, two insert expressions (i1, i2), two update expressions (u1, u2) and two delete expressions (d1,d2) were edited. Figure 11 presents the source code to configure RBTC. Each CRUD expression is deployed to the associated Business Tier Entity type. From now on, they reside in a pool and may be re-used by application tiers whenever necessary. Figure 12 presents the source code to implement the case shown in Figure 1. There is no need to master the database schema, there is no need to write any CRUD expression and there is no need to write source code for the business tier part. RBTC provides schema-driven and

```
74      // Table Prd_products
75      int Prd_productId() throws SQLException;
76      String Prd_productName() throws SQLException;
77      int PrdSup_supplierID() throws SQLException;
78      int PrdCat_categoryId() throws SQLException;
79      String Prd_quantityPerUnit() throws SQLException;
80      float Prd_unitPrice() throws SQLException;
81      float Prd_unitsInStock() throws SQLException;
```

Figure 9. Example of an IRead interface.

[1] http://dl.dropbox.com/u/71192544/Work/Confers/ICIS/ICIS_2013/RBTC.7z

```
70      // Table Prd_products
71      void Prod_productName(String productName)
72                          throws SQLException;
73      void PrdSup_supplierID(int supplierID)
74                          throws SQLException;
75      void PrdCat_categoryId(int categoryId)
76                          throws SQLException;
77      void Prd_quantityPerUnit(String quantityPerUnit)
78                          throws SQLException;
79      void Prd_unitPrice(float unitPrice)
80                          throws SQLException;
81      void Prd_unitsInStock(float unisInStock)
82                           throws SQLException;
```

Figure 10. Example of an IWrite interface.

```
93  void config(IConfig config) throws BTC_Exception {
94      config.addCRUD(1,s1,RBTC.crudType.crud_s);
95      config.addCRUD(2,s2,RBTC.crudType.crud_s);
96      config.addCRUD(1,u1,RBTC.crudType.crud_u);
97      config.addCRUD(2,u2,RBTC.crudType.crud_u);
98      config.addCRUD(1,i1,RBTC.crudType.crud_i);
99      config.addCRUD(2,i2,RBTC.crudType.crud_i);
100     config.addCRUD(1,d1,RBTC.crudType.crud_d);
101     config.addCRUD(2,d2,RBTC.crudType.crud_d);
102 }
103
```

Figure 11. Configuration process of RBTC.

```
130 void updUnitPrice(ISession session,
131                    List<Integer> productID,
132                    List<Float> unitPrice)
133                    throws SQLException,
134                           BTC_Exception {
135     ISelect s=session.createBTW_S(1,
136                       RBTC.scrollabilty.forwardOnly,
137                       RBTC.updatability.updatable);
138     for (int n=0; n<productID.size(); n++) {
139         s.execute(new Object[]{productID.get(n)});
140         if (s.hasNext()) {
141             pn=s.Prd_productName();
142             supID=s.PrdSup_supplierID();
143             // ... more code
144             uPrice=unitPrice.get(n);
145             s.Prd_unitPrice(uPrice);
146             s.updateRow();
147             // ... more code
```

Figure 12. Example shown in Figure 1 but based on a component derived from the RBTA.

```
152 void updUnitsInStock(ISession session,
153                       List<Integer> productId,
154                       List<Float> units)
155                       throws SQLException,
156                              BTC_Exception {
157     ISelect s=session.createBTW_S(1,
158                       RBTC.scrollabilty.forwardOnly,
159                       RBTC.updatability.updatable);
160     for (int n=0; n<productId.size(); n++) {
161         s.execute(new Object[]{productId.get(n)});
162         if (s.hasNext()) {
163             pn=s.Prd_productName();
164             supID=s.PrdSup_supplierID();
165             // ... more code
166             uInStock=units.get(n);
167             s.Prd_unitsInStock(uInStock);
168             s.updateRow();
169             // ... more code
```

Figure 13. Example shown in Figure 2 but based on a component derived from the RBTA.

type-safe methods to access each attribute, and CRUD expressions are selected from a pool. CRUD expressions are easily used and reused as shown in Figure 13, which implements the case shown in Figure 2. Here, once again, there is no need to master database schemas in spite of using an additional attribute (line 167) and no need to re-write any CRUD expression or need to re-write source-code as happened in Figure 2. The counterpart to obtain these advantages is limited to the effort of writing an IRead interface and the required CRUD expressions.

VI. Conclusion and Future Work

CLI are tools used to build business tiers whenever a fine tune control on the interactions with host databases is a key requirement. In spite of this relevant advantage, four drawbacks of CLI were emphasized. To tackle these drawbacks, and simultaneously keep CLI advantages, an architecture has been proposed: RBTA. A proof of concept based on JDBC has been presented. From the proof of concept, the following conclusions are taken: 1) Programmers are relieved from mastering database schemas. This advantage of RBTA comes from its schema-driven and type-safe approach for the signatures of all getter and setter methods. 2) Programmers do not need to re-write the same CRUD expression more than once. RBTA provides a pool service through which programmers manage the availability and the use of CRUD expressions. 3) Programmers do not need to go through the traditional and manual maintenance activities of source code of business tiers. If new CRUD expressions are needed (or updated) and their CRUD schemas are within the implemented Business Tier Interface, then no maintenance is required. If the new CRUD schemas are not within the implemented Business Tier Interface, then there is the need to add (update) the new attributes to the IRead interface and automatically rebuild the new component. 4) Unlike CLI and other O/RM tools, RBTA provides an environment where business tiers and applications tiers are completely decoupled. The counterpart to achieve these advantages is the need to write and maintain an IRead interface and submit it to a tool responsible for building RBTC, which is an automated process. Another key advantage of RBTA is that RBTC, unlike CLI, transform runtime errors into compile errors. If the name of an attribute is modified, a new RBTC is statically built. Then, when the application tier is re-compiled, the compiler will detect all errors where the source-code of application tiers was not updated. With CLI, names of attributes are encoded inside strings, this way preventing any disconformity from being detected at compile time.

In order to evaluate the possibility of using other technologies than CLI, an RBTC was manually built in C#, ADO.NET and SQL Server 2008. The achieved success proved that RBTA is flexible enough to be used with different technologies. From our experience with O/RM tools, namely Java Persistence API, it is our belief that RBTA may also be used. However, RBTA is so easy to be used with CLI that it would only bring disadvantages if used with O/RM tols, namely because of their induced overhead.

In future work, we plan to build a tool to help the building and maintaining processes of IRead interfaces. Basically it reads the schema of database tables and proposes the methods to be contained by IRead. Additionally, it will be possible to define methods not directly derived from the database schema in order to support user defined attributes.

References

[1] M. David, "Representing database programs as objects," Advances in Database Programming Languages, F. Bancilhon and P. Buneman, eds., pp. 377-386, N.Y.: ACM, 1990.

[2] W. Cook, and A. Ibrahim, "Integrating programming languages and databases: what is the problem?," 2011 May: ODBMS.ORG, Expert Article, 2005.

[3] M. Parsian, JDBC Recipes: A Problem-Solution Approach, NY, USA: Apress, 2005.

[4] G. T. Heineman, and W. T. Councill, Component-Based Software Engineering: Putting the Pieces Together, 1st ed., pp. 880: Addison-Wesley, 2001.

[5] C. Szyperky, D. Gruntz, and S. Murer, Component Software - Beyond Object-Oriented Programming: Addison-Wesley/ACM Press, 2002.

[6] L. Kung-Kiu, and W. Zheng, "Software Component Models," IEEE Trans. on Soft. Eng., vol. 33, no. 10, pp. 709-724, 2007.

[7] A. Bracciali, A. Brogi, and C. Canal, "A formal approach to component adaptation," Journal of Systems and Software, vol. 74, no. 1, pp. 45-54, 2005.

[8] P. V. Elizondo, and K.-K. Lau, "A Catalogue of Component Connectors to Support Development with Reuse," Journal of Systems and Software, vol. 83, no. 7, pp. 1165-1178, 2010.

[9] Microsoft. "Northwind and pubs Sample Databases for SQL Server 2000.," 2012; http://www.microsoft.com/download/en/details.aspx?id=23654.

[10] R. C. William, and R. Siddhartha, "Safe query objects: statically typed objects as remotely executable queries," in 27th Int. Conf. on Software Engineering, St. Louis, MO, USA, 2005, pp. 97-106.

[11] Oracle. "Java Data Objects (JDO)," 2011 Nov; http://www.oracle.com/technetwork/java/index-jsp-135919.html.

[12] A. M. Russell, and H. K. Ingolf, "SQL DOM: compile time checking of dynamic SQL statements," in 27th Int. Conf. on Software Engineering, St. Louis, MO, USA, 2005, pp. 88-96.

[13] J. L. Gregor Kiczales, Anurag Mendhekar, Chris Maeda, Cristina Lopes Videira, Jean-Marc Loingtier, Joh Irwin, "Aspect-Oriented Programming," in ECOOP, Jyvaskyla,Finland, 1997, pp. 220-242.

[14] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, Greenwich,CT,USA: Manning Publications, 2003.

[15] J. Fabry, and T. D'Hondt, "KALA: Kernel Aspect Language for Advanced Transactions," in Proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France, 2006, pp. 1615-1620.

[16] T. Dinkelaker, "AO4SQL: Towards an Aspect-Oriented Extension for SQL," in 8th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'11), Zurich, Switzerland, 2011, pp. 1-5.

[17] Microsoft. "Microsoft Open Database Connectivity," Jul, 2012; http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx.

[18] G. Mead, and A. Boehm, ADO.NET 4 Database Programming with C# 2010, USA: Mike Murach & Associates, Inc., 2011.

[19] Oracle. "ResultSet," 2012 Jul; http://docs.oracle.com/javase/6/docs/api/java/sql/ResultSet.html.

[20] Microsoft. "RecordSet (ODBC)," 2011 Jun; http://msdn.microsoft.com/en-us/library/5sbfs6f1.aspx.

[21] Microsoft. "ADO.NET DataSet," Dec, 2011; http://msdn.microsoft.com/en-us/library/zb0sdh0b(v=vs.80).aspx.